APPENDIX

We present PARALLELKITTENS performance on Blackwell GPUs (Appendix A), additional collective performance results (Appendix B), PARALLELKITTENS API specification (Appendix C), program template and example kernels (Appendix D), multi-GPU setup process (Appendix E), in-network acceleration setup process (Appendix F), and the artifact appendix for reproducing results (Appendix G).

A BLACKWELL GPU PERFORMANCE

In this section, we demonstrate that PK generalizes across different hardware architectures by presenting representative kernel performance on Blackwell GPUs and comparing against available baselines that also support this architecture.

All experiments were conducted using 8×Nvidia B200 GPUs, interconnected via 5th-generation NVLink and NVSwitch (900 GB/s unidirectional bandwidth), using CUDA 12.8 and PyTorch 2.8.0. All matrix multiplications use FP32 as the tensor core accumulator type. For brevity, we denote the GEMM shape as $M \times N \times K$, where the first operand has dimensions $M \times K$ and the second has dimensions $K \times N$. We report the observed average compute throughput.

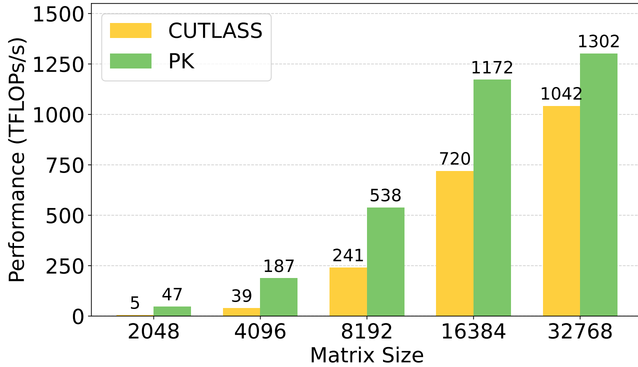

Figure 13. BF16 AG + GEMM performance. Local GEMM size is $N \times N/8 \times N$, with N given on the X-axis.

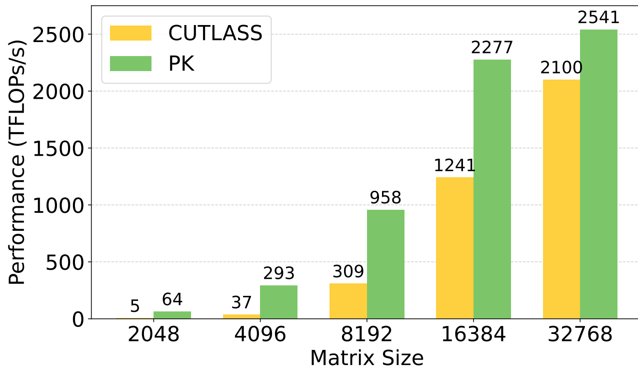

Figure 14. FP8 AG + GEMM performance. Local GEMM size is $N \times N/8 \times N$, with N given on the X-axis.

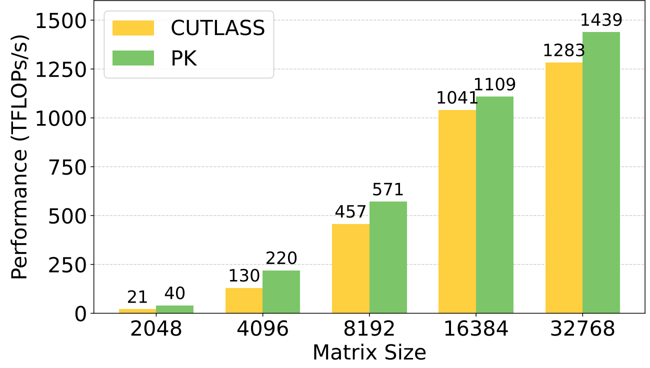

Figure 15. BF16 GEMM + RS performance. Local GEMM size is $N \times N \times N/8$, with N given on the X-axis.

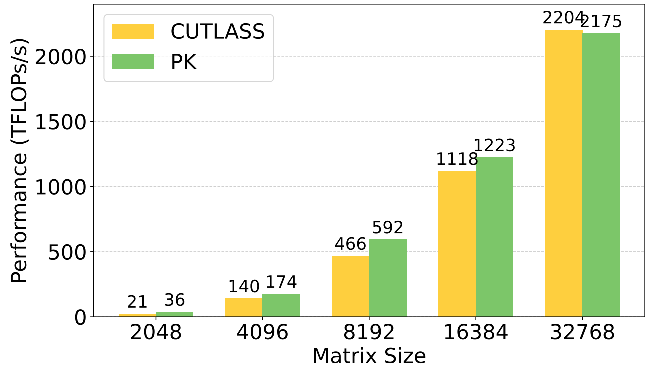

Figure 16. FP8 GEMM + RS performance. Local GEMM size is $N \times N \times N/8$, with N given on the X-axis.

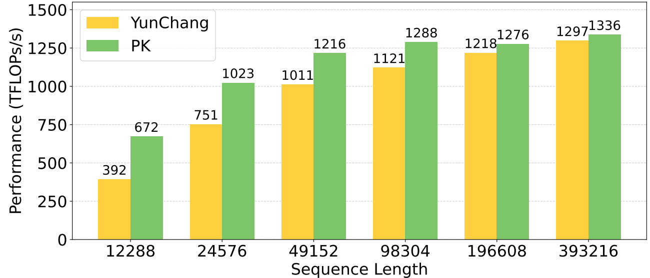

Figure 17. BF16 DeepSpeed-Ulysses attention layer performance across sequence lengths ($B = 16$, $H = 128$, $D = 128$).

B ADDITIONAL COLLECTIVE PERFORMANCE

In this section, we report additional results on pure collective kernel performance and compare them against NCCL. We particularly examine how performance can improve significantly when the communication pattern is *fine-grained*: for example, when performing all-gather or reduce-scatter along the tensor dimension (the last dimension) instead of the batch dimension (the first dimension), or when performing all-to-all operations across head and sequence dimensions. In such cases, the memory layout becomes discontiguous,

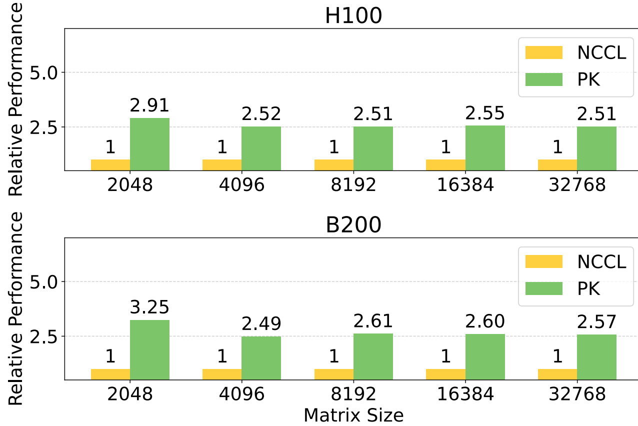

Figure 18. BF16 tensor dimension all-gather performance comparison. The gathered matrix size is $N \times N$, with N given on the X-axis.

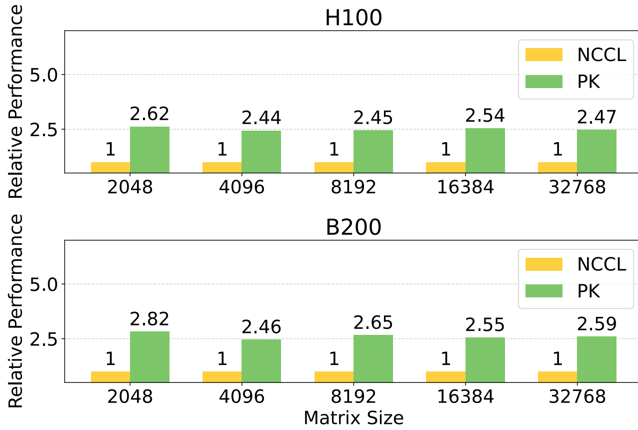

Figure 19. BF16 tensor dimension reduce-scatter performance comparison. The scattered matrix size is $N \times N/8$, with N given on the X-axis.

which makes NCCL inefficient, as it supports collectives only on contiguous partitions and thus requires extra reshaping and copying. In contrast, PK can execute these collectives directly on the original layout. The results below illustrate this advantage.

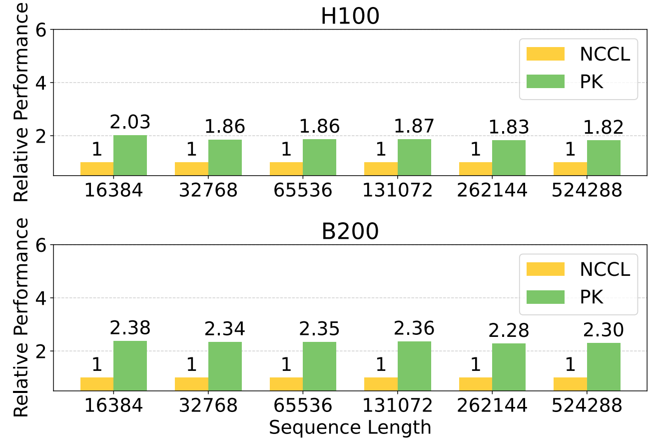

Figure 20. 4-dimensional (B, S, H, D) BF16 all-to-all performance comparison, with $B = 1$, $H = 128$, $D = 128$, and varying S given on the X-axis. The S dimension is gathered and the H dimension is evenly scattered across 8 GPUs.

C PARALLELKITTENS API SPECIFICATION

We provide the full specification of PK primitives, including each function's name, signature, parameters, and description.

store_async

```
template <
    int axis,
    cache_policy policy,
    kittens::ducks::st::all ST,
    kittens::ducks::pgl::all PGL,
    kittens::ducks::coord::tile COORD>
__device__ void store_async(
    const PGL &dst,
    const ST &src,
    const COORD &idx)
```

Template Parameters:

- `axis`: Tensor axis for the operation (0-3).
- `policy`: Cache policy (NORMAL or cache hint).
- `ST`: Shared tile type.
- `PGL`: Parallel global layout type.
- `COORD`: Coordinate type for indexing.

Parameters:

- `dst`: Destination parallel global layout.
- `src`: Source shared memory tile.
- `idx`: Coordinate specifying the destination position.

Description: Asynchronously stores a shared memory tile to multicast memory using the Tensor Memory Accelerator (TMA). Launched by a single thread.

store_add_async

```
template <
    int axis,
    cache_policy policy,
    kittens::ducks::st::all ST,
    kittens::ducks::pgl::all PGL,
    kittens::ducks::coord::tile COORD>
__device__ void store_add_async(
    const PGL &dst,
    const ST &src,
    const COORD &idx)
```

Template Parameters:

- `axis`: Tensor axis for the operation (0-3).
- `policy`: Cache policy (NORMAL or cache hint).
- `ST`: Shared tile type.
- `PGL`: Parallel global layout type.
- `COORD`: Coordinate type for indexing.

Parameters:

- `dst`: Destination parallel global layout.
- `src`: Source shared memory tile.
- `idx`: Coordinate specifying the destination position.

Description: Asynchronously performs an atomic add reduction from a shared memory tile to multicast memory via TMA. The operation atomically adds the source tile values to the existing values at the destination. Launched by a single thread.

`reduce`

```
template <
    int TILE_ROWS,
    int TILE_COLS,
    kittens::reduce_op OP,
    kittens::ducks::pgl::all PGL,
    kittens::ducks::gl::all GL>
__device__ void reduce(
    GL &dst,
    const coord &dst_idx,
    PGL &src,
    const coord &src_idx)
```

Template Parameters:

- `TILE_ROWS`: Number of rows in the tile.
- `TILE_COLS`: Number of columns in the tile.
- `OP`: Reduction operation to apply (sum, max, or min).
- `PGL`: Parallel global layout type.
- `GL`: Global layout type.

Parameters:

- `dst`: Reference to the destination global layout.
- `dst_idx`: Coordinate specifying the destination tile's position.
- `src`: Reference to the source parallel global layout.
- `src_idx`: Coordinate specifying the source tile's position.

Description: Performs a reduction from multicast memory to device-local global memory. The function loads data from the source parallel global layout using in-network reduction operations and stores the result to the destination global layout. Collectively launched by one or more warps. Each warp processes multiple rows of the tile, performing the specified reduction operation during the multicast load and then writing the reduced values to the destination global memory.

`all_reduce`

```
template <
    int TILE_ROWS,
    int TILE_COLS,
    kittens::reduce_op OP,
    kittens::ducks::pgl::all PGL>
__device__ void all_reduce(
    PGL &dst_and_src,
    const coord &idx)
```

Template Parameters:

- `TILE_ROWS`: Number of rows in the tile.
- `TILE_COLS`: Number of columns in the tile.
- `OP`: Reduction operation to apply (sum, max, or min).
- `PGL`: Parallel global layout type.

Parameters:

- `dst_and_src`: Reference to the parallel global layout object.
- `idx`: Coordinate specifying the tile's position with batch (`b`), depth (`d`), row (`r`), and column (`c`) indices.

Description: Performs an all-reduce collective operation on a tile of data on multicast memory. The function reduces data across all participating GPUs for the specified tile. Collectively launched by one or more warps. Each warp processes multiple rows, loading data from multicast memory with the specified reduction operation, then writing the result back to the same multicast location. The operation leverages in-network acceleration hardware to efficiently perform the reduction without explicit peer-to-peer copies.

`signal`

```
__device__ void signal(
    const barrier_t &barrier,
    const coord &idx,
    const int dst_dev_idx,
    const int val)
```

Parameters:

- `barrier`: Reference to the barrier object (parallel global layout of integers).
- `idx`: Element-wise coordinate specifying the barrier location.
- `dst_dev_idx`: Target device index to signal.

- `val`: Value to add to the barrier counter.

Description: Signals a specific device's barrier by atomically adding a value to its counter. This primitive is used to coordinate synchronization between thread blocks and GPUs.

`signal_all`

```
__device__ void signal_all(
    const barrier_t &barrier,
    const coord &idx,
    const int val)
```

Parameters:

- `barrier`: Reference to the barrier object.
- `idx`: Element-wise coordinate specifying the barrier location.
- `val`: Value to add to all devices' barrier counters.

Description: Signals all devices simultaneously by performing a multicast atomic add operation. Uses in-network multicast hardware to efficiently update barrier counters across all participating devices with a single operation.

`wait`

```
__device__ void wait(
    const barrier_t &barrier,
    const coord &idx,
    const int dev_idx,
    const int expected)
```

Parameters:

- `barrier`: Reference to the barrier object.
- `idx`: Element-wise coordinate specifying the barrier location.
- `dev_idx`: Device index to wait on.
- `expected`: Expected barrier value to wait for.

Description: Waits until a device's barrier counter reaches the expected value. Continuously polls the barrier location using relaxed memory ordering loads until the expected value is observed. This provides a spinning wait mechanism for inter-SM and inter-GPU synchronization.

`barrier`

```
__device__ void barrier(
    const barrier_t &barrier,
    const coord &idx,
    const int dev_idx)
```

Parameters:

- `barrier`: Reference to the barrier object.
- `idx`: Element-wise coordinate specifying the barrier location.
- `dev_idx`: Current device index.

Description: Implements a complete barrier synchronization across all devices. This ensures all participating GPUs reach the same synchronization point before proceeding.

D PARALLELKITTENS PROGRAM TEMPLATE AND EXAMPLE KERNELS

Load-Compute-Store-Communicate (LCSC) Template. The LCSC template provides a structured approach for implementing multi-GPU kernels with specialized worker components. The template enables flexible warp/SM specialization and overlapping strategies for compute, memory, and communication operations.

High-level Template Structure:

```
struct lcsc_template {
    static void loader(
        globals, comp_sem,
        comp_smem, comp_regs);
    static void storer(
        globals, comp_sem,
        comp_smem, comp_regs);
    static void consumer(
        globals, comp_sem,
        comp_smem, comp_regs);
    static void communicator(
        globals, comm_sem,
        comm_smem, comm_regs);
};
```

Required Components:

- `comp_sem`: struct of semaphores for synchronization within compute SMs.
- `comm_sem`: struct of semaphores for synchronization within communication SMs.
- `comp_smem`: struct of shared memory layouts for compute SMs.
- `comm_smem`: struct of shared memory layouts for communication SMs.
- `comp_regs`: struct of register state for compute workers.
- `comm_regs`: struct of register state for communication workers.

Workers:

- `loader`: Performs memory loads from local or peer HBM using TMA.
- `storer`: Performs memory stores to local or peer HBM.

```
__device__ inline void loader(const globals &G, comp_sem &sem, comp_smem &smem, comp_regs &regs) {
    int2 idx = interpret_task(regs.task_id);
    for (int red_idx = 0; red_idx < regs.num_iters; red_idx++) {
        wait(sem.inputs_finished[regs.stage], get_phasebit<1>(regs.phasebits, regs.stage));
        update_phasebit<1>(regs.phasebits, regs.stage);
        tma::expect_bytes(sem.inputs_arrived[regs.stage], sizeof(A_tile) * 2 + sizeof(B_tile));
        if (red_idx == PIPELINE_STAGES - 1) {
            wait(sem.outputs_finished, get_phasebit<1>(regs.phasebits, PIPELINE_STAGES));
            update_phasebit<1>(regs.phasebits, PIPELINE_STAGES);
        }
        for (int i = 0; i < 2; i++)
            tma::load_async(smem.inputs[regs.stage].A[i], G.A, {idx.x * 2 + i, red_idx}, sem.inputs_arrived[regs.
                ↪ stage]);
        tma::load_async(smem.inputs[regs.stage].B, G.B, {red_idx, idx.y}, sem.inputs_arrived[regs.stage]);
        regs.stage = (regs.stage + 1) % PIPELINE_STAGES;
    }
}

__device__ inline void storer(const globals &G, comp_sem &sem, comp_smem &smem, comp_regs &regs) {
    int2 idx = interpret_task(regs.task_id);
    wait(sem.outputs_arrived, get_phasebit<0>(regs.phasebits, 0));
    update_phasebit<0>(regs.phasebits, 0);
    for (int i = 0; i < 2; i++)
        tma::store_async(G.C[G.dev_idx], regs.C[i], {idx.x * 2 + i, idx.y});
    tma::store_async_read_wait();
    arrive(sem.outputs_finished);
    int signal_dev_idx = regs.task_id % NUM_DEVICES;
    device<NUM_DEVICES>::signal(G.barrier, {idx.x, idx.y}, signal_dev_idx, 1);
}

__device__ inline void consumer(const globals &G, comp_sem &sem, comp_smem &smem, comp_regs &regs) {
    rt_fl<ROW_BLOCK / 8, COL_BLOCK> C_accum;
    warp::zero(C_accum);
    for (int red_idx = 0; red_idx < regs.num_iters; red_idx++) {
        wait(sem.inputs_arrived[regs.stage], get_phasebit<0>(regs.phasebits, regs.stage));
        update_phasebit<0>(regs.phasebits, regs.stage);
        warpgroup::mma_AB(C_accum, smem.inputs[regs.stage].A[regs.warpgroup_id], smem.inputs[regs.stage].B);
        warpgroup::mma_async_wait();
        warp::arrive(sem.inputs_finished[regs.stage]);
        regs.stage = (regs.stage + 1) % PIPELINE_STAGES;
    }
    group<8>::sync(3);
    warpgroup::store(regs.C[regs.warpgroup_id], C_accum);
    warpgroup::sync(regs.warpgroup_id + 1);
    warpgroup::arrive(sem.outputs_arrived);
}

__device__ inline void communicator(const globals &G, comm_sem &sem, comm_smem &smem, comm_regs &regs) {
    int2 idx = interpret_task(regs.task_id);
    if (threadIdx.x == 0)
        device<NUM_DEVICES>::wait(G.barrier, {idx.x, idx.y}, G.dev_idx, NUM_DEVICES);
    __syncthreads();
    group<NUM_WARPS>::all_reduce<ROW_BLOCK, COL_BLOCK, reduce_op::ADD>(G.C, {idx.x, idx.y});
}
```

Figure 21. Fused GEMM + AR kernel implemented with the LCSC template

- `consumer`: Performs tensor/CUDA core operations on loaded data.
- `communicator`: Performs dedicated inter-GPU communication. Executes on separate communication SMs.

Execution Model: The template automatically distributes SMs between computation and communication roles based on `num_comm_sms`, passed in to the host entry function. Compute SMs execute loader, storer, and consumer functions with producer-consumer synchronization through semaphores. Communication SMs execute the communicator function independently. The framework handles warpgroup specialization, register allocation, and task distribution across workers. Programmers can utilize this template by defining the above struct, and passing it to the launch interface:

```
lcsc::launch_kernel<config,
    globals, lcsc_template>(G, stream);
```

Where the parameters are:

- `config`: Compile-time configuration struct defining SM and thread counts.
- `globals`: Runtime globals struct containing device memory pointers and parameters.
- `lcsc_template`: User-defined LCSC template implementation.
- `G`: Instance of globals struct.
- `stream`: CUDA stream for kernel execution.

We present a fused GEMM + all-reduce (AR) kernel implemented using the LCSC template in Figure 21. This exem-

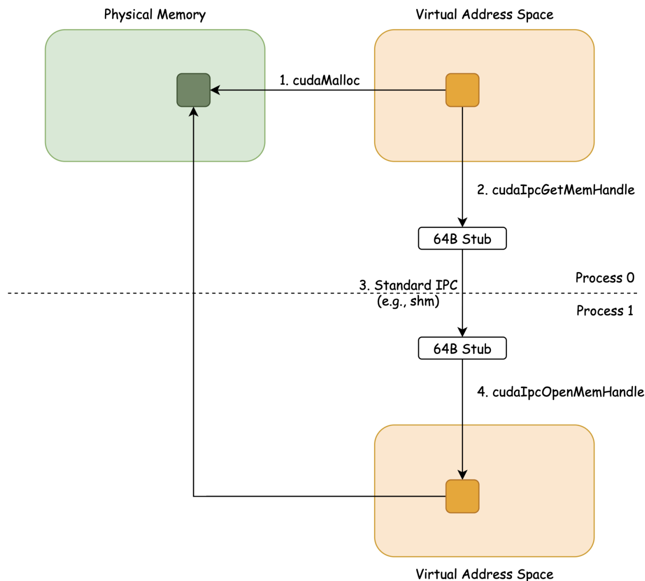

Figure 22. CUDA IPC flow.

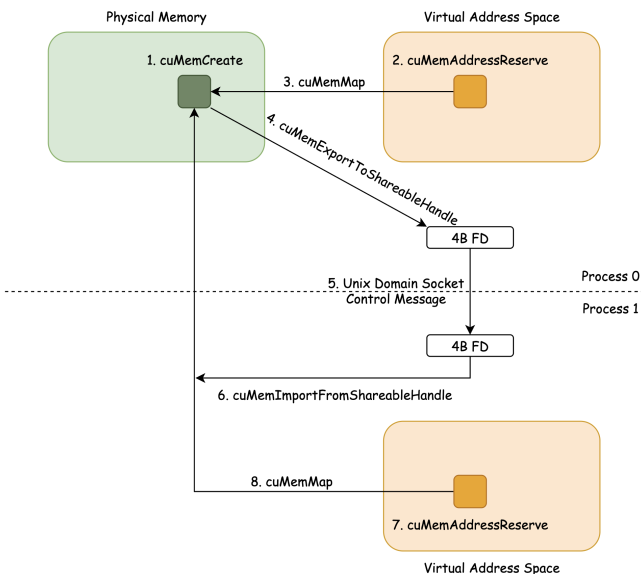

Figure 23. CUDA VMM flow.

plifies how the LCSC template decomposes a computation-communication overlapping kernel into four specialized workers using an inter-SM overlapping scheme. The `loader` iterates over the reduction (K) dimension, issuing asynchronous loads of A and B tiles from local HBM into shared memory and signaling the arrival after each tile so tensor-core MMAs can proceed. The `consumer` waits on these signals, performs block-level MMA on each tile, accumulates results across K, and after completing accumulation, writes the output tile to shared memory and signals the storer. The `storer` waits for the completed output tile, writes it to local HBM, and signals the communicator. The `communicator` waits until all GPUs have stored their corresponding tile, then performs an all-reduce across GPUs on the matching output locations.

We highlight that the kernel contains *both* a fully optimized GEMM and fused all-reduce logic, with the communication-relevant code comprising only about 10 lines of device code. We also open-source all remaining kernels evaluated in this paper through our GitHub repository.

E MULTI-GPU MEMORY SETUP PROCESS

We describe the low-level multi-GPU memory setup process, a major complexity in multi-GPU programming, which PK abstracts away from programmers.

The basic requirement of multi-GPU programming is that kernels must be able to access memory (HBM) on peer devices. To enable this, we need to create a new mapping in the current device's virtual address space that points to the peer device's physical memory. After that, the kernel can simply dereference the address, and the NVLink and NVSwitch fabric handle the underlying transfer.

There are three ways to create such mappings: (1) CUDA Unified Virtual Addressing, (2) CUDA Inter-Process Communication, and (3) manual Virtual Memory Management.

E.1 CUDA Unified Virtual Addressing (UVA)

UVA provides a single unified virtual address space across GPUs, but with the limitation that it applies only within a single process. That is, if we avoid using multiple processes altogether, there exist no heterogeneous virtual address spaces.

However, we note that modern production distributed training and inference are built around a multi-processing model. Distributed runners like `torchrun` assume 1 GPU device per rank (process), and working around this is quite complicated. Thus, multi-processing is the preferred model of launching multi-GPU workloads, which brings us to the next two methods.

E.2 CUDA Inter-Process Communication (IPC)

Calling `cudaIpcGetMemHandle` on the address in the current virtual address space returns a 64-byte stub that can be shared across processes through standard IPC mechanisms like shared memory or Unix domain sockets. The receiving process then can call `cudaIpcOpenMemHandle`, which maps the given stub into its own address space. Figure 22 visualizes this flow.

While this method is straightforward and works on pre-allocated device memory (e.g., existing PyTorch tensors), its drawback is that it cannot use the NVSwitch accelerator for faster reduction and broadcast operations.

E.3 Manual Virtual Memory Management (VMM)

For VMM, we start by manually allocating the GPU physical memory with `cuMemCreate`. This allows setting the `CU_MEM_HANDLE_TYPE_POSIX_FILE_DESCRIPTOR` property on this physical memory, which then lets us export the physical memory reference as a Linux file descriptor by calling `cuMemExportToShareableHandle`.

Because file descriptors are tied to a specific process in Linux, they cannot be shared directly. The standard way to transfer a file descriptor in Linux is to send it as a control message over a Unix domain socket. Once we send the file descriptor over to the destination process, it can then import the physical memory reference using `cuMemImportFromShareableHandle` and map it into its own virtual address space using the VMM API. The overall flow is illustrated in Figure 23.

A downside of this approach is that the given memory must be allocated with VMM and is subject to size granularity requirements, typically at 2MB for H100s and B200s. As a result, a PyTorch-allocated tensor, which is usually allocated by the standard cudaMalloc without size alignment, cannot be shared directly across processes. Instead, we need a custom tensor class that manages device memory allocation and deallocation with custom VMM logic. The main advantage, however, is that this method enables the use of NVSwitch in-network accelerators.

F IN-NETWORK ACCELERATION SETUP PROCESS

In order to utilize NVSwitch acceleration, we first allocate local memory on each participating device with VMM. Then we create a *multicast object*, which is an abstraction over multiple physical locations in multiple devices. To do this, we create an 8-byte stub that represents the multicast object with `cuMulticastCreate`, register all devices as participants, and map each device's physical memory to it.

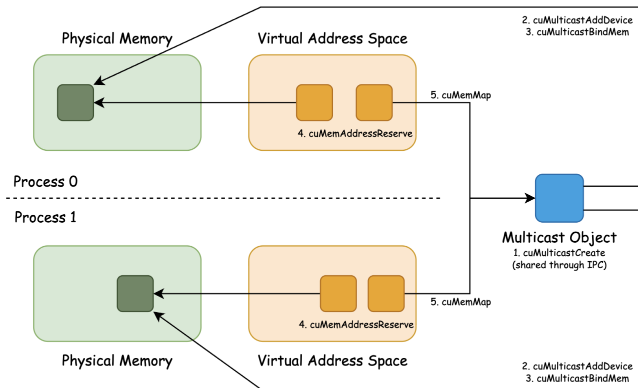

Figure 24. CUDA multicast object creation process.

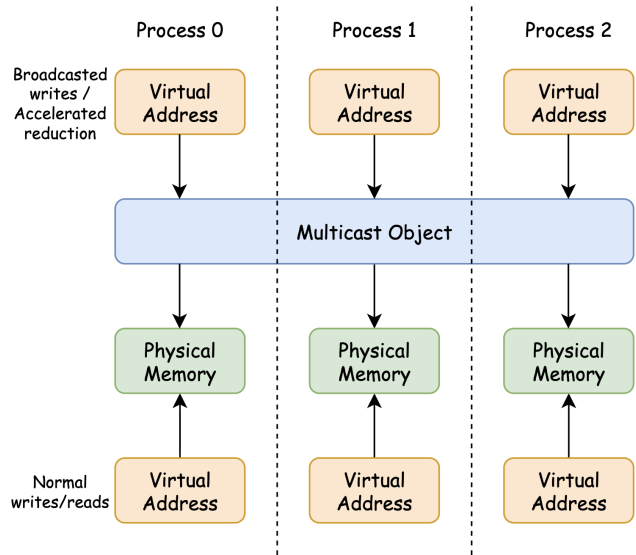

Figure 25. CUDA multicast object hierarchy.

A multicast object behaves just like VMM-allocated physical memory: we can share it with other processes and map a virtual address to it using the same mechanism described in the VMM setup process. That is, we export the multicast object as a POSIX file descriptor, open it on each device, and map it into each process's virtual address space. The overall setup process and the exact names of the CUDA functions called are shown in Figure 24.

After completing the above, each process has two addresses: one mapping to the current device's physical memory (local address) and another mapping to the multicast object (multicast address). Writing to and reading from the local address is a standard global memory access. Writing to the multicast address triggers a broadcast across all participating devices, multicasted in the NVSwitch fabric. Reading from the multicast address causes undefined behavior. Finally, in-fabric reduction operations can be invoked on the multicast address using the PTX instructions `multimem.red` and `multimem.ld_reduce`. This is illustrated in Figure 25.

G ARTIFACT APPENDIX

G.1 Abstract

The PARALLELKITTENS artifact provides the source code and scripts needed to reproduce the throughput results reported in Sec. 4 of the main paper. The artifact consists of PK-based CUDA C++ kernels compiled as PyTorch extensions, a build system, and Python benchmark scripts. Each benchmark runs on a single node using multi-GPU intra-node communication. It reports throughput in TFLOP/s and, optionally, numerical correctness against a reference implementation.

G.2 Artifact check-list (meta-information)

- **Program:** ParallelKittens multi-GPU kernels.
- **Compilation:** GNU Make with GNU C++ and `nvcc` (CUDA 12.8).
- **Binary:** Generated locally using the provided Makefile (`make`).
- **Dataset:** Synthetic random tensors generated at runtime.
- **Run-time environment:** Ubuntu 22.04 or 24.04, Nvidia GPU driver compatible with CUDA 12.8, CUDA Toolkit 12.8, Python 3.12, PyTorch 2.8.0 compatible with CUDA 12.8, `pybind11` 3.0.1.
- **Hardware:** Single node with 8×H100 GPUs connected via NVLink/NVSwitch.
- **Execution:** Launched by running `make run` in the corresponding directory.
- **Metrics:** Compute throughput (TFLOP/s).
- **Output:** Console logs on `stdout`.
- **Experiments:** Reproduction of PK throughput reported in Figures 7–12 (Sec. 4).
- **How much disk space required (approximately)?:** Approximately 70 MB, excluding the CUDA installation and the benchmark script's Python dependencies.
- **How much time is needed to prepare workflow (approximately)?:** Approximately 15 minutes.
- **How much time is needed to complete experiments (approximately)?:** Approximately 30 minutes.
- **Publicly available?:** Yes (GitHub).
- **Code licenses (if publicly available)?:** MIT license.
- **Archived (provide DOI)?:** Yes (https://doi.org/10.5281/zenodo.19458048).

G.3 Description

G.3.1 How delivered

The artifact is delivered as a public GitHub repository: https://github.com/HazyResearch/ThunderKittens. The relevant directory for this paper is `ThunderKittens/kernels/parallel/`. Specifically, the PK kernels required to reproduce the results in Sec. 4 are located in:

- `kernels/parallel/ag_gemm`
- `kernels/parallel/gemm_rs`
- `kernels/parallel/gemm_ar`
- `kernels/parallel/ring_attn`
- `kernels/parallel/ulysses_attn`
- `kernels/parallel/moe_dispatch_gemm`

G.3.2 Hardware dependencies

This artifact should be evaluated on a single node with 8 Nvidia H100 SXM5 80 GB GPUs connected via 4th-generation NVLink/NVSwitch.

G.3.3 Software dependencies

The following software dependencies must be satisfied before compiling and executing the artifact:

- OS: Ubuntu 22.04 or 24.04.
- CUDA Toolkit 12.8.
- Environment variables `PATH` and `LD_LIBRARY_PATH` set to the corresponding `bin/` and `lib64/` paths in the local CUDA directory, respectively.
- Python 3.12.
- PyTorch 2.8.0 built for CUDA 12.8.
- Pybind11 3.0.1.
- GNU Make.

G.3.4 Data sets

No external datasets are required. All benchmarks use synthetic random tensors generated at runtime.

G.4 Installation

First, ensure that CUDA Toolkit 12.8 is installed on the system. Instructions for downloading the CUDA Toolkit are available on the official Nvidia website: https://developer.nvidia.com/cuda-12-8-0-download-archive.

Then, follow these instructions. Although we use Miniconda3 for the Python environment, this is not required; users may use another virtual environment system (e.g., `venv`) or a system-wide Python installation.

G.4.1 Install Miniconda

```
mkdir -p ${HOME}/miniconda3
wget https://repo.anaconda.com/miniconda/Miniconda3-
     latest-Linux-x86_64.sh -O ${HOME}/miniconda3/
     miniconda.sh
bash ${HOME}/miniconda3/miniconda.sh -b -u -p ${HOME}/
     miniconda3
rm ${HOME}/miniconda3/miniconda.sh

source ${HOME}/miniconda3/bin/activate
conda init --all
conda create --name pk python=3.12 -y
conda activate pk
```

G.4.2 Install Python dependencies

```
pip install torch==2.8.0 --index-url https://download.
     pytorch.org/whl/cu128
pip install pybind11==3.0.1
```

```
# For Figures 10 and 11 only:
pip install "flash-attn-4==4.0.0b7"
```

G.4.3 Set up CUDA

```
export PATH=/usr/local/cuda/bin:$PATH
export LD_LIBRARY_PATH=/usr/local/cuda/lib64:
    $LD_LIBRARY_PATH
```

G.4.4 Set up ThunderKittens

```
git clone https://github.com/HazyResearch/
    ThunderKittens

PYTHON_VERSION=$(python3 -c "import sysconfig; print(
    sysconfig.get_config_var('LDVERSION'))")
PYTHON_INCLUDES=$(python3 -c "import sysconfig; print('-
    I', sysconfig.get_path('include'), sep='')")
PYTHON_LIBDIR=$(python3 -c "import sysconfig; print('-L
    ', sysconfig.get_config_var('LIBDIR'), sep='')")
PYBIND_INCLUDES=$(python3 -m pybind11 --includes)
PYTORCH_INCLUDES=$(python3 -c "from torch.utils.
    cpp_extension import include_paths; print(' '.join
    (['-I' + p for p in include_paths()]))")
PYTORCH_LIBDIR=$(python3 -c "from torch.utils.
    cpp_extension import library_paths; print(' '.join
    (['-L' + p for p in library_paths()]))")

export PYTHON_VERSION="${PYTHON_VERSION}"
export PYTHON_INCLUDES="${PYTHON_INCLUDES}"
export PYTHON_LIBDIR="${PYTHON_LIBDIR}"
export PYBIND_INCLUDES="${PYBIND_INCLUDES}"
export PYTORCH_INCLUDES="${PYTORCH_INCLUDES}"
export PYTORCH_LIBDIR="${PYTORCH_LIBDIR}"
export GPU=H100
```

G.5 Experiment workflow

The following workflow reproduces the PK throughput reported in Figures 7–12. Each directory under `kernels/parallel/` contains a standalone benchmark. After completing the installation steps described above, compiling and running each experiment is as simple as changing into the corresponding directory and running `make run`. For each benchmark, start from the repository root and run:

1. **Figure 7 (AG + GEMM):**

```
cd kernels/parallel/ag_gemm
make run
```

2. **Figure 8 (GEMM + RS):**

```
cd kernels/parallel/gemm_rs
make run
```

3. **Figure 9 (GEMM + AR):**

```
cd kernels/parallel/gemm_ar
make run
```

4. **Figure 10 (Ring Attention):**

```
cd kernels/parallel/ring_attn
make run
```

5. **Figure 11 (DeepSpeed-Ulysses Attention):**

```
cd kernels/parallel/ulysses_attn
make run
```

6. **Figure 12 (Token Dispatch + GEMM):**

```
cd kernels/parallel/moe_dispatch_gemm
make run
```

G.6 Evaluation and expected result

Each `make run` command compiles the PK kernels into a PyTorch extension and runs an 8-process `torchrun` benchmark using the provided Python benchmark script, printing timing and throughput to `stdout`. The `TK ... | ... TFLOP/s` lines correspond to PK kernel performance.

The benchmark scripts include an optional correctness mode that compares the PK kernel output against a reference implementation. For a quick correctness sanity check, set `check_correctness=True` for a small problem size in the corresponding `benchmark.py` and re-run the benchmark; the script will print an error summary. Note that small numerical differences consistent with BF16 arithmetic are expected.